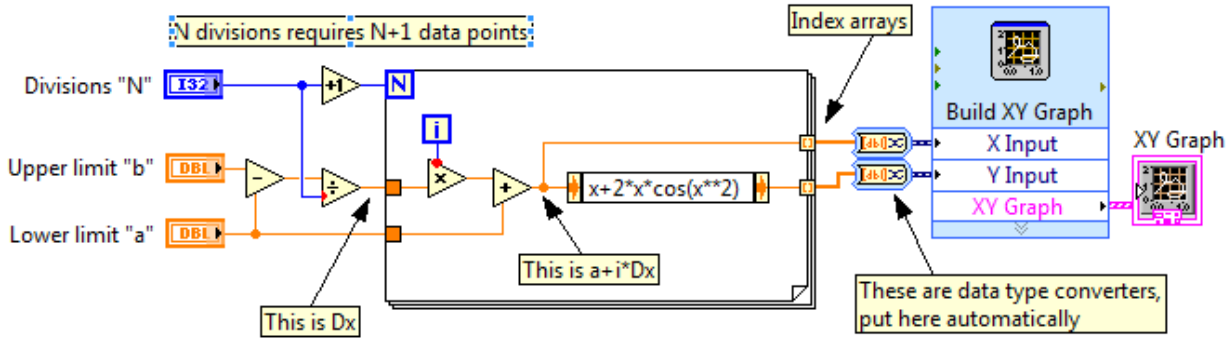
Worked Example: Function Generation

To graph a function $y=f(x)$, generate a set of x and corresponding y values, and then send them to the *Express XY Graph* for display. To sequentially generate the x values, consider starting at a supplied lower bound $x = a$, and then stepping forward one point at a time by a specified increment Δx to a final upper bound $x = b$. We might choose to specify a certain number of uniform steps or divisions N to use instead, such that:

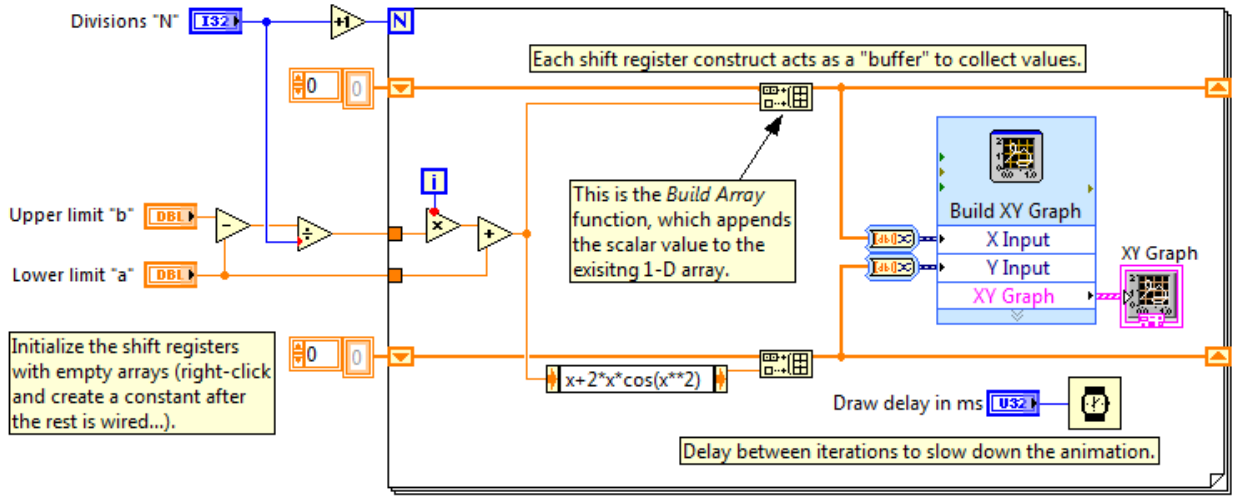
$$\Delta x \equiv \frac{b-a}{N}$$

Thus, all of the x values are generated as $x_i = a + i \cdot \Delta x$, for $i = \{0,1,2,3,\ldots,N\}$. This process is easily implemented in an iterated loop, in which the $f(x)$ values are also evaluated within the loop at the same time. If the x and $f(x)$ values are indexed within the loop, then two arrays are generated containing all of the x and corresponding $f(x)$ values for the graph. Create a VI to implement this procedure for the function $f(x) = x + 2x \cdot \cos(x^2)$, as shown:



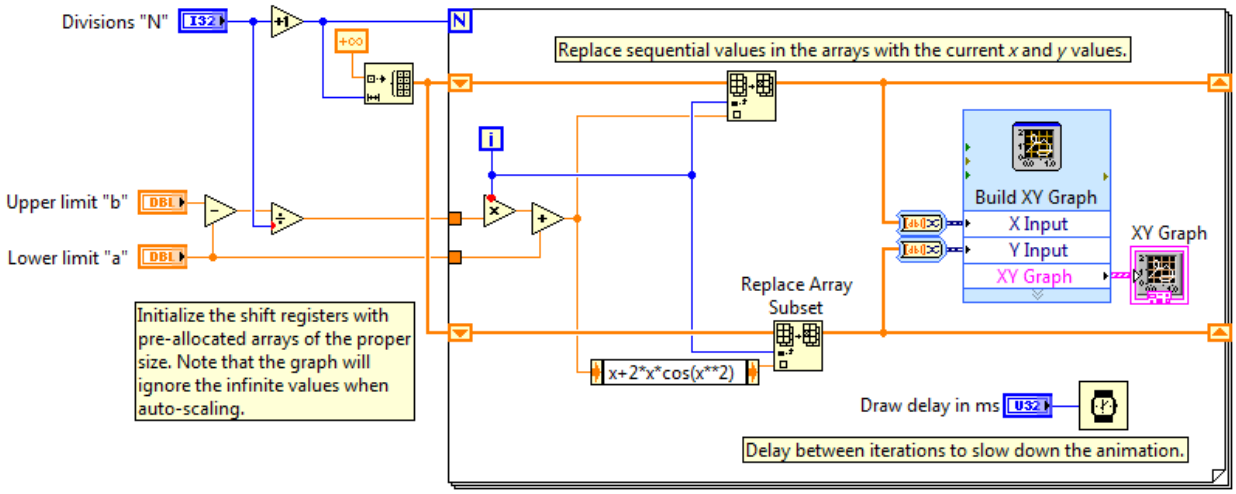
Note that no shift registers are needed here, since none of the values are recalled from previous iterations. This is the benefit of constructing the x values as $x_i = a + i \cdot \Delta x$. Also note that the graph is drawn only at the conclusion of the loop (which happens very quickly).

What if we wanted to see the graph draw as the values are being generated instead, producing an interesting animation of the function? We cannot simply move the graph into the loop without some re-programming, because only one x-y pair is currently generated at a time, so the graph will not display properly. So, we would need to create "growing arrays" that sequentially increase in size as the loop progresses. This approach would require the use of shift registers as shown:



The shift registers act as a "buffer" to collect the points for display. The empty arrays that initialize the shift registers are very important here, since they enable the graph to start over from the beginning when the VI is re-executed. If they were removed, the re-draw effect would not work properly, and more importantly, the arrays would continue to get larger with every run of the VI.

As it turns out, even as shown, the alternate approach above is not as efficient as it could be. The *Build Array* function forces LabVIEW to re-allocate memory for the growing arrays, which takes extra resources from the processor. An even better approach is to pre-allocate the size of the arrays (which we known based on *N*), and then sequentially replace elements as the loop iterates:



No memory re-allocation is necessary for this, which speeds up the processing. Although we aren't really interested in speeding things up in this example, other scenarios that need buffers and speed can benefit greatly.